John von Neumann Institute for Computing

A Distributed Scheme for Fault-Tolerance in Large Clusters of Workstations

A. Duarte, D. Rexachs, E. Luque

published in

Parallel Computing:
Current & Future Issues of High-End Computing,
Proceedings of the International Conference ParCo 2005,
G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E. Zapata (Editors),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **33**, ISBN 3-00-017352-8, pp. 473-480, 2006.

http://www.fz-juelich.de/nic-series/volume33

A distributed scheme for fault-tolerance in large Clusters of Workstations *†

Angelo Duarte[ab], Dolores Rexachs[a], Emilio Luque[a]

[a]Computer Architecture and Operating Systems Department, University Autonoma of Barcelona, ETSE, Edifici Q, Bellaterra, 08193, Barcelona, Spain

[b]Computer Science Department, University Catholic of Salvador, Av. Cardeal da Silva, 205, Federacão, 40220-140, Salvador, Bahia, Brazil

1. Introduction

Fault tolerance is a critical point for long-running parallel-distributed applications executing in Massive Cluster of Workstations (MCOW). From the user's point of view, a parallel application should run and finish correctly, but users rarely want to worry about including fault tolerance capabilities in their algorithms because of the software engineering costs. The cluster's administrators, in turn, request that the fault tolerance scheme consume as few resources as possible. Because increasing the number of nodes causes the MTBF to reduce, long-running applications demands a fault tolerance scheme that be independent of the cluster scalability.

The fault tolerance scheme could rely on a fault-tolerant hardware; however, such solution is expensive in practice. An alternative would be to develop fault-tolerant algorithms. However, such solution demands a big software engineering effort, and it cannot be applied to algorithms already coded. A third possibility is to build a software layer between the application and the system in order to isolate the faults from the application. This is an interesting alternative since it does not require any special hardware, and makes the fault tolerance scheme fully transparent to the user.

Rollback-recovery is the classical method used when it is necessary to offer fault tolerance for a long-running parallel-distributed application based on message passing in a cluster [7]. Rollback-recovery techniques can be implemented by a software layer, and are divided into two broad categories: the ones based only in checkpoints and the ones based in checkpoint and in message logs. The first ones make checkpoints of individual processes associated with a certain checkpoint synchronization scheme to assure that the system rolls back and recover from a consistent global state after a failure. The latter ones also aggregate message's logs for each individual process in order to allow the system to recover from a point later than the latter consistent global state [5].

The main difference between these techniques is expressed by two parameters: a) the runtime overhead over fault-free execution and b) the fault penalty. The overhead directly relates to the efficiency of the rollback-recovery scheme without faults, and it is strongly related to the cluster resources consumed by the rollback-recovery protocol [4]. For example, from the user's point of view, a fault tolerance scheme using message log interferes on the latency of message's transmission. Furthermore, both checkpoints and logs require a additional storage resources and such resources impact over the cluster architecture.

The fault penalty depends on the efficiency of the rollback-recovery scheme after a failure. It derives from two main parameters: a) the cost of recovering itself and b) the cluster architecture

*This work was supported by the MCyT-Spain under contract TIC 2004-03388 and partially supported by the Generalitat de Catalunya - Grup Recerca Consolidat 2001 SGR-00218

†The first author is thankful to the Universidade Catolica do Salvador (UCSal), Brasil, for the financial support during this work

after the failure. The cost depends on how far a set of processes must rollback in order to reach a consistent global state, i.e., how much computation is lost. Factors like process's interaction and checkpoint interval strong influence the cost of recovering. For example, short checkpoint intervals permit that a process loses little computation because it needs to recover from a relatively recent point. However, short checkpoint intervals strongly influence the system performance because of the frequent interruptions in the process's execution. The cluster architecture after a failure, in turn, determines how many nodes are available to continue the computation.

Several studies have focused on the behavior of the rollback-recovery protocols and some implementations have been built for practical systems (see section 2.5). Such studies and implementations have concentrated on the particularities of the protocols or on the performance aspects, but they gave little attention to the relationship between the fault tolerance scheme and the cluster architecture.

Such relationship is important because it allows an evaluation of the impact of the fault tolerance scheme over the cluster architecture. Furthermore, using such relationship it is possible to build models for the cluster architecture in the presence of failures. These models are very useful when the user needs to know how many failures his/her system can bear, or how failures interfere on his/her application.

Taking into consideration that any fault tolerance scheme for MCOWs must be scalable and that the relationship between the cluster architecture and the fault tolerance scheme is relevant in order to allow the user to evaluate its application in the presence of failures, we have developed a fault tolerant architecture that attends to both requisites.

Our architecture RADIC bases on two distributed arrays of dedicated processes that together implement a distributed fault tolerance controller. One array is composed by *protector* processes, which are in charge of the cluster's nodes, and the other array is composed by *observer* processes, which care of the processes of the parallel-distributed application. Both arrays of processes work transparently and independently in order to isolate faults from the application. Such architecture establishes a deterministic behavior for the cluster architecture after a node failure and it is easily adaptable to the cluster configuration.

The next section contains a description of the architecture, and a comparison with some related works. Section 3 presents the validation of the architecture using a practical implementation. In section 4, we state our conclusions and relate the future works.

2. RADIC: A scalable architecture for fault tolerance in clusters

RADIC - Redundant Array of Distributed Independent Checkpoints is a functional architecture model based on two arrays of system processes: *protectors* and *observers*. Together, these processes compose a distributed fault tolerance controller.

Protectors are processes that monitor each cluster node, and function like a distributed stable storage system. *Observers* are processes that control the checkpoints and message logs of each application process (one *observer* for each application process).

Figure 1a depicts the interaction between processes in a cluster with four nodes (N1..N4). Each node has a protector process (T1..T4). There are five *observer* process (Oa..Oe), each one attached to an application processes (A..E). Doted lines indicate the relations between *protectors* processes, and continuous lines indicate the relations between *observers* and *protectors*.

The overhead caused by RADIC over a failure-free operation is the same caused by any fault-tolerance scheme since it does not add new overheads for the recovery mechanism. The overhead on application runtime will mainly depend on three factors: a) runtime enlargement of the application processes caused by the checkpoints, b) message delaying caused by the logs and c) computation of

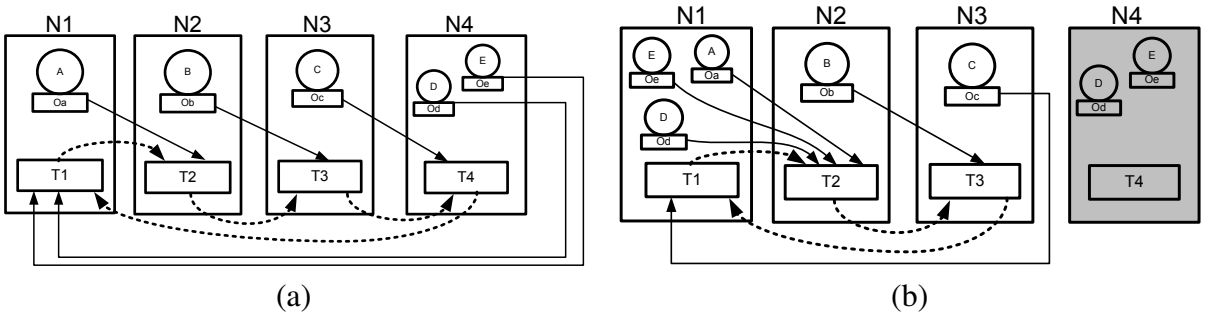

Figure 1. An application composed by 5 process (A..E) running in a cluster with 4 nodes (N1..N4) including the RADIC components. a) Operation without failures; b) A fault in N4 forces T3 to find T1, Oc connects to protector T1, and T1 recovers processes D and E.

the recover protocol algorithm. The cluster suffers a performance reduction because of: a) the increasing of disk I/O (checkpoints and logs storage) in each machine and b) the increasing of network traffic (checkpoint and log transmission from the *observers* to *protectors*).

Fault penalty depends on: a) the number of processes that will rollback; b) how much computation is lost in each application process rollback; and c) the final architecture of the cluster after a failure. The last factor increases application runtime if the final cluster architecture has fewer nodes, which causes a lower performance. Nevertheless, in a large scale cluster, loss of just a few nodes in the cluster will probably have little influence on overall performance.

In the next sections we explain the basic functionality of RADIC, a more detailed explanation about the RADIC architecture functionality and modules can be found in [6].

2.1. System Model

A distributed application consists of a set of concurrent executing processes that cooperate with each other to perform a task. The processes communicate only through message passing. There are P processes in a cluster with N nodes. In failure-free executions, all the N nodes are available. The system will support a Maximum Number of Failure Nodes (MNFN). If a node fails, it will be definitely discarded. Processes that were placed in a faulty node always recover in a different node.

The distributed application does not interact with the outside world. Therefore, received messages are the only nondeterministic event and each process is modeled as a sequence of state intervals, each one started by a received message. Execution inside each interval is deterministic.

Communication channels and process are both synchronous, i.e., whenever an element is working correctly, it always will perform its intended function in a finite and known (or predictable) time bound. Therefore, every communication channel will have a latency bound and every process will execute each of their state intervals in a time bound.

2.2. *Protectors* processes

Each *protector* communicates with another *protector* in a different node, in such a way that every *protectors* is monitored by some other *protectors*. Together, all *protectors* perform a distributed failure detector. When a protector is performing a monitoring function, it sets a watchdog for each *protector* it is monitoring. The monitored *protector* regularly sends control messages in order to reset the watchdog of its monitor. If a failure occurs in a monitored node, the watchdog of the monitor *protector* detects it, and the monitor *protector* starts the necessary actions in order to recover the application processes that were placed in the faulty node. Similarly, if a failure occurs in a monitor node, each monitored *protector* detects it because it cannot send the reset message to its monitor's watchdog. In such situation, the monitored *protector* connects to a new monitor.

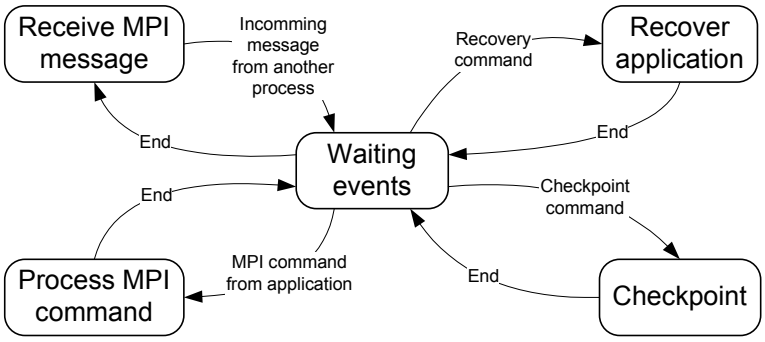

Figure 2. *Observer* state diagram

Each *protector* also communicates to a set of *observers* that are in the nodes that it monitors. For each *observer* of its set, the *protector* operates as a remote stable storage; i.e., the *protector* stores the checkpoints (and message logs) of the application processes related to each *observer* of its set. In Figure 1a, *protector* T4 monitors T3, and is monitored by T1 (a similar scheme is followed by the other *protectors*). Therefore, T4 stores the checkpoints and message logs of the application process C (via Oc) and T1 stores the checkpoints and message logs of application processes D and E (via Od and Oe).

Different *protector*'s interconnections schemes are possible. For example, the nodes can be grouped in cells regarded to protection scheme, where each cell is built by a chain of *protectors* as depicted in Figure 1a. Another possibility is to use nodes only to protect (nodes with no application process) and nodes only to compute (nodes with no protector process).

2.3. *Observers* processes

Observers are RADIC processes attached to each application processes. Each *observer* process is "owned" by an application process and has to perform several different tasks. The first task is to manage the message passing between its application process and the other processes. The second task is to maintain a mapping table indicating the location of all application processes and their respective *protectors*, i.e., in which node each application process and its respective *protector* is located. This table is updated whenever an *observer* detects a communication failure with another application process in the MPI world. Each *observer* uses a simple heuristic in order to update its table: if the communication fails, look for the process reincarnation in the node of monitor protector of the faulty process. Such heuristic avoids that the new location of a faulty node needs to be transmitted for every non-faulty node in the cluster. The third task is taking checkpoints and message-logs of its application process, and send them to the monitor protector. For non-coordinated schemes, each *observer* can have an individual checkpoint policy for its application process. Such independence allows the implementation of efficient checkpoint strategies in order to reduce the overhead caused by checkpoints.

Finally, each *observer* is responsible for performing the rollback-recovery activities when the system uses fault tolerance schemes that demands coordination during recovery. Is such cases, the *protectors* determine the which specific checkpoint is necessary for each process in order to roll back the system to a consistent state. Such checkpoints are "send back" to the respective *observer*, and the *observer* manages the roll back of their application processes.

Figure 2 shows the *observer* state diagram. The state transitions are fired by events that comes from three elements: its monitoring protector, other MPI processes and the application process related to the *observer*. There are two different events related to the monitoring protector. The first

event occurs when the *observer* takes checkpoints and message logs from its application process and sends them to the monitoring protector of the node where the *observer* is placed.

The second event occurs when the *observer* receives a recover command from the protector indicating that it should restart the application process from a previous state (for rollback recovery protocols that requires coordination).

The *observer* also has events related to other application processes in the MPI world, whenever they send messages to its application process. Finally, there are events related to application process of the *observer* whenever this process performs an MPI command.

It should be noted that for protocols that need to keep several checkpoints for each application process, the *observer* also must to maintain a copy of each checkpoint sent to the *observer*. Such requirement comes because the monitoring protector itself is unreliable. Therefore, if the monitoring protector fails the checkpoint history of their monitored processes is lost and hereafter such processes cannot be recovered.

2.4. Failure detection and recovery

Figure 1b represents a failure in node N4. In this case, T3 (monitored) and T1 (monitor) detect the failure in T4. T3 connects to monitor and finds T1. Meanwhile, because T1 was monitoring N4, T1 recovers the application processes of the faulty node N4. Simultaneously, the *observers* in the node N3 (in this case only Oc) detect that its checkpoint storage (protector T4) has failed, and search a new protector in order to establish a new checkpoint storage for their application processes.

Before a protector recovers an application process, it first determines the correct checkpoint that should be used. For protocols that require a coordinated recovery, the protector array starts a synchronization procedure in order to roll back the system to a consistent state. Since each protector has checkpoints of a subset of the application processes, they command the *observers* to resume their application processes from an earlier state in order to reach a system-global consistent state.

In order to achieve this, each protector transmits the specific checkpoint back to the respective *observer*, and commands the *observer* that restarts the application process from this checkpoint. For message log protocols, besides rolls back the system to a consistent state, the protector also replays the messages that are in the message log.

The synchronization between *protectors* is necessary only for protocols that require global coordination. The message logging pessimistic protocol does not require such coordination because the recovery data and the recovery protocol itself can rely only in local information [7].

2.5. Comparison with other solutions

There are many solutions using rollback-recovery for implementing fault tolerance in parallel-distributed systems dedicated to execute scientific long-running applications. Projects such as Starfish, CoCheck, Egida, FT-MPI, MPI-FT, LAM-MPI, MPICH-V, LA-MPI and Open MPI represent some recent efforts in attempting to incorporate network and process fault tolerance into message passing systems using checkpoint and rollback-recovery techniques.

Although such projects do present valid solutions to the fault tolerance problem in clusters, they focus on performance issues or on protocol details. They dedicate little or none attention to questions about how the cluster architecture interacts with the fault tolerance scheme or how failures influence the cluster architecture.

The RADIC architecture simultaneously attend to the following requisites: scalability, transparency, modeling of the relationship between fault-tolerance scheme and the cluster architecture, and do not request any dedicated nodes in order to operate. Therefore, in this section we have compared RADIC and other solutions taking these requisites in consideration.

CoCheck [13] relies on the Condor checkpoint library, and it is implemented on the top of tuMPI. It uses coordinated checkpoint and the recover is based on a centralized coordinator. Starfish [1] provides failure detection and recovery based on coordinated or uncoordinated checkpoint. Starfish lets the responsibility of recovering to the application. Egida [11] is a toolkit integrated with MPICH. It changes the *p4* parallel programing library send/receive functionalities, and was dedicated to compare the behavior of different rollback-recovery protocols. FT-MPI [8] is not transparent to the application. It only handles failures at the MPI communicator and the application must manage the recovery. MPI-FT [10] uses message logging together with a centralized pessimistic strategy based on a central *observer* or a distributed optimistic strategy based on each application process. In case of a failure, MPI-FT restarts a faulty process since the beginning. MPICH-V [3] does define the architecture for the cluster configuration. However, its fault-tolerant scheme relies in dedicated nodes to achieve its goal. LAM/MPI [12] uses the Berkeley Labs Checkpoint Library in order to implement a coordinated checkpoint protocol. However, it does not offer an automatic mechanism to failure detection and recovery. LA-MPI [2] focuses on network fault tolerance and does not offer fault tolerance for the application processes. Open MPI [9] is a recent MPI-2 compliant project that include fault tolerance capabilities in their implementations. Open MPI is a combination of the technologies from FT-MPI, LA-MPI, LAM-MPI and PACX-MPI. At the time this text is written, fault tolerance is still not available as a stable feature.

3. Architecture Validation

We validate the RADIC functionality using a prototype implementation called RADICMPI. This implementation includes a library (*radicmpi*) and a runtime environment (*mpicc* and *mpirun*) that facilitates the compilation and the program executions.

The current implementation of the library *radicmpi* contains the following subset of MPI functions: MPI_Init, MPI_Finalize, MPI_Send, MPI_Recv, MPI_Comm_rank, MPI_Comm_size, MPI_Get_processor_name, MPI_Wtime, MPI_Type_size. RADICMPI uses the pessimistic message log rollback-recovery protocol because it is the only that does not compromise the system scalability since it confines the effects of a failure only to the faulty process. Furthermore, this protocol simplifies garbage collection because the system can simply discard checkpoints and message logs previous to the most recent checkpoint.

Using RADICMPI, we tested the functionality of RADIC with two programs: ping-pong and matrix multiplication. Our main interest was to confirm the architecture functionality in the presence of failures. We used the matrix multiplication program in order to certify the system correctness under failure conditions. The ping-pong program was used to verify the functionality of each RADIC module and to control the injection of failures.

We ran the tests in a heterogeneous cluster with six nodes interconnected by a 100BaseT hub: 3 Athlon XP 2600+/1.9GHz/256MB (w2,w3,w4); 1 Pentium 4/2.6GHz/256MB (w5); and 2 Pentium-III/800MHz/128MB (master,w1). All nodes used Linux Fedora Core 3 with kernel 2.6.9-1.667. For checkpointing we used the library developed by Victor Zandy [14]. All softwares were compiled using GNU g++ compiler v3.4.2. The *protectors* array was organized in a chain like Figure 1a.

The overheads for a master-worker matrix multiplication algorithm are summarized in Figure 3. In order to see the impact of logs over the overall runtime, we used an algorithm that sends one matrix for all workers and then slice the other matrix among the workers. First, we run the program without any kind of protection (checkpoints or logs). Then, we executed the algorithm with all protections activated and forcing a checkpoint approximately at the middle of the execution. One can note the strong impact of logs and checkpoints over the overall performance. This impact is caused by the increasing in the message latency caused by message logging mechanism. Furthermore, the

	master	w1	w2	w3	w4	w5
250x250	141	56	133	123	104	65
500x500	58	27	67	62	50	33
1000x1000	32	41	12	11	8.3	5.4

Figure 3. RADIC runtime overheads for a matrix multiplication algorithm. The results were calculated based on a execution without protection (overhead=runtime protected/runtime unprotected).

large memory space used by the program leads to a large checkpoint overhead because of the large checkpoint storage time caused (hard disks in the protectors). The RADIC efficiency improves when the matrix increase in size because the reduction of the log impact in the whole computation time. Since we were interested only in the functionality of the RADIC architecture, we did not make any performance measure for these cases because the total application runtime after a failure depends on the moment where the failure occurs inside a checkpoint interval, because this moment determines how much a process rolls back.

4. Conclusions

We presented and described RADIC, an architecture model for implementing fault tolerance in clusters based on the concept of *observer* and *protector* processes. RADIC covers all the requirements of a fault tolerance architecture for parallel-distributed systems, and also have important features. It is scalable since it implements a fully distributed scheme for supporting faults. It is user transparent since it does not impose any change to the application algorithm. It is independent of the recover protocol, because the networks of *protectors* can store checkpoints and message logs from the *observers* in order to attend to the different strategies. It should be noted that RADIC also allow the implementation of non-scalable rollback-recovery protocols, like the ones that require global coordination in order to perform recovery. Furthermore, non-transparent fault-tolerant can also take advantage of the RADIC architecture.

We have proved the basic functionalities of the RADIC architecture making tests with RADICMPI. Now, we are developing a performance model for evaluating the total application runtime as a function of parameters like checkpoint interval; checkpoint cost; message patterns; application algorithm; RADIC organization; failure pattern; and computation/communication ratio. Our interest is to investigate how the cluster architecture is influenced by failures. Our intention is to build a model in order to allow that the user either evaluates the impact of the fault-tolerance scheme over his/her application or determine which cluster architecture should attend to his/her application runtime requisites.

We continue the development of RADICMPI in order to make it more efficient for measuring the parameters necessary for building and validating the performance model. We are also improving RADICMPI in order to use it with the NAS benchmark. Furthermore, we are evaluating the viability of using RADIC with implementations like OpenMPI or MPICH2. Such possibility would greatly facilitate new experiments.

We are also interested in evaluating how the different RADIC configurations operates in massive clusters. Since scalability is one of our main goals, we are interested in studying how the efficiency of the protection is affected by the number of machines in the cluster. Furthermore, since massive clusters can be constructed with different types of network topology, we are concerned about methods for distributing the *observers* taking into consideration their distances to the *protectors*.

References

[1] A.M. Agbaria and R. Friedman. Starfish: fault-tolerant dynamic MPI programs on clusters of workstations. In *Proc. of 8th Inter. Symp. on High Perf. Dist. Computing*, pages 167–176, August 1999.

[2] R.T. Aulwes, D.J. Daniel, N.N. Desai, R.L. Graham, L.D. Risinger, M.A. Taylor, T.S. Woodall, and M.W. Sukalski. Architecture of LA-MPI.

[3] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. MPICHV: Toward a scalable fault tolerant MPI for volatile nodes. In *Proc. of SuperComputing 2002 (SC2002)*, November 2002.

[4] A. Bouteiller, P. Lemarinier, G. Krawezik, and F. Cappello. Coordinated checkpoint versus message log for fault tolerant MPI. In *In Proc. of 2003 IEEE International Conference on Cluster Computing*, pages 242–250. IEEE, December 2003.

[5] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[6] A.A. Duarte. RADIC: Redundant array of distributed independent checkpoints. Master's thesis, Universidad Autónoma de Barcelona, Departamento de Arquitectura de Computadores y Sistemas Operativos, ETSE, Bellaterra, 08193, Barcelona, Spain, July 2005.

[7] E.N. Elnozahy, L. Alvisi, Y.M. Wang, and D.B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[8] G. Fagg and J. Dongarra. FT-MPI: Fault tolerant mpi, supporting dynamic applications in a dynamic world. In *Euro PVM/MPI User's Group Meeting 2000*, pages 346–353, Berlin, Germany, 2000. Springer-Verilag.

[9] E. Gabriel, G.E. Fagg, G. Bosilca, and et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proc., 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[10] S. Louca, N. Neophytou, A. Lachanas, and P. Evripidou. MPI-FT: Portable fault tolerance scheme for MPI. *Parallel Processing Letters*, 10(4):371–382, 2000.

[11] S. Rao, L. Alvisi, and H. Vin. Egida: An extensible toolkit for low-overhead fault-tolerance. In *Proc. of IEEE Fault-Tolerant Computing Symposium (FTCS-29)*, Madison, WI, June 1999.

[12] S. Sankaran, J.M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. In *Proc. of LACSI Symposium*, Sante Fe, New Mexico, USA, October 2003.

[13] G. Stellner. Cocheck: Checkpointing and process migration for MPI. In *Proc. of Inter. Parallel Processing Symp.*, pages 526–531, Los Alamitos, CA, USA, April 1996. IEEE Computer Society Press.

[14] V. Zandy. Ckpt - a process checkpoint library. http://www.cs.wisc.edu/ zandy/ckpt/, April 2005.